Two-enqueuer queue in Common2

David Eisenstat*

Abstract

The question of whether all shared objects with consensus number 2 belong to Common2, the set of objects that can be implemented in a wait-free manner by any type of consensus number 2, was first posed by Herlihy. In the absence of general results, several researchers have obtained implementations for restricted-concurrency versions of FIFO queues. We present the first Common2 algorithm for a queue with two enqueuers and any number of dequeuers.

1 Introduction

Many concurrent algorithms employ first-in first-out (FIFO) queues, making the quality of queue implementations by particular synchronization primitives a practical concern. In this work, we restrict our attention to **wait-free** implementations, where processes cannot take infinitely many steps without completing one of their operations. Wait-freedom is an especially strong fault-tolerance property, ensuring that processes make progress despite contention and unexpected delays; unsurprisingly, there are a number of impossibility results regarding wait-free implementations. Many of these follow from the consensus hierarchy of Herlihy [8], who defined the **consensus number** of a data type T to be the least upper bound on all n such that an n-process system with some collection of objects of type T or Register can implement consensus. Since the composition of wait-free simulations is wait-free, no type can implement a type with a higher consensus number. For example, Register, which has consensus number 1, cannot implement Queue, which has consensus number 2.

However, the consensus hierarchy does not let us determine the structure of the "can implement" relation for types with the same consensus number. Herlihy [8] showed that in an n-process system, any type with consensus number $n' \geq n$ is **universal**, that is, it can implement all types. He asked whether Fetch&Add, which has consensus number 2, can implement all types with consensus number 2 in systems with three or more processes. Several researchers have found implementations for specific types, but as of this writing, neither a universal implementation nor a counterexample is known.

*No current affiliation. 55 Autumn Street, New Haven, CT 06511, USA. eisenstatdavid@gmail.com

Table 1: Summary of known wait-free queue implementations from a type of consensus number 2 in an n-process system

Enqueuers	Dequeuers	Distinct values	References
1	n	arbitrary	David [4, 5]
n	2	arbitrary	Li [10]
n	n	1	David, Brodsky, and Fich [6]
2	n	arbitrary	this work

Afek, Weisberger, and Weisman showed that any type with consensus number 2 can implement Fetch&Add [2, 3] and Swap [3, 11].[1] They defined Common2 to be the set of types that can be implemented by any type of consensus number 2. Afek, Gafni, and Morrison [1] showed that Stack is in Common2, improving on an implementation for two pushers by David, Brodsky and Fich [6]. The status of Queue remains unknown, however, despite the existence of several restricted implementations. When all enqueue operations have the same argument, Queue and Stack have the same specification, and the one-value Stack implementation by David, Brodsky, and Fich [6] is also a one-value Queue implementation. Li [10] obtained an implementation for multiple values and one dequeuer from an algorithm by Herlihy and Wing [9]. He extended it to two dequeuers via the universal implementation technique and conjectured that there is no three-dequeuer implementation. David [4, 5] refuted this conjecture by giving an implementation for one enqueuer and any number of dequeuers, observing, however, that its enqueue operation is not amenable to the same technique. We describe a variant of David's algorithm that admits a two-enqueuer extension, leaving open the case of three enqueuers and three dequeuers. The known queue implementations are summarized in Table 1.

Given that modern architectures typically offer a primitive of consensus number ∞, our implementation is of mainly theoretical interest, though we believe that it contributes to a better understanding of the synchronization required to implement Queue. For this reason, we have not attempted to reduce the space requirements of our algorithms.

2 Model

The setting for this work is the standard asynchronous shared-memory model. We describe this model only informally; the interested reader should consult a formal description such as the one by Herlihy [8].

A **shared-memory system** consists of n sequential **processes** and a collection of shared (base) **objects**. Processes communicate with other processes by performing operations on the objects. Each object has a **type**, which specifies the sequential behavior of the methods that it supports as functions from an object state to a return value and a new state. Table 2

[1] In turn, Fetch&Add can implement all read-modify-write (RMW) types with commuting updates, and Swap can implement all RMW types with overwriting updates.

Table 2: Types used in this paper

Type	Consensus number	Method	Defining function Obj. State → Return Val. × Obj. State
Consensus	∞	decide(x)	$y \mapsto \begin{cases} (x, x) & \text{if } y = \bot \\ (y, y) & \text{if } y \neq \bot \end{cases}$
Fetch&Add	2	f&a(x)	$y \mapsto (y, y + x)$
Queue	2	deq()	$\langle\rangle \mapsto (\bot, \langle\rangle)$ $\langle x \rangle \circ q' \mapsto (x, q')$
		enq(x)	$q \mapsto (\text{Ok}, q \circ \langle x \rangle)$
Register	1	read()	$y \mapsto (y, y)$
		write(x)	$y \mapsto (\text{Ok}, x)$
Stack	2	pop()	$\langle\rangle \mapsto (\bot, \langle\rangle)$ $s' \circ \langle x \rangle \mapsto (x, s')$
		push(x)	$s \mapsto (\text{Ok}, s \circ \langle x \rangle)$
Swap	2	swap(x)	$y \mapsto (y, x)$

$\langle \cdots \rangle$ denotes a sequence. $\circ$ denotes concatenation. $\bot$ is a return value that indicates failure. Ok indicates success in the absence of a value to return.

lists each type used in this paper along with its consensus number, the methods that it supports, and their defining functions. A **schedule** is an arbitrary sequence of processes; in the wait-free setting, there are no fairness conditions. Each schedule gives rise to an **execution**, where starting from some initial state, the processes take **steps** according to the schedule. When a process takes a step, it selects an operation based on the return values of past operations and performs it atomically.

In order to reason about wait-free implementations, we augment the base objects with a virtual object of the type being implemented. Whenever a process attempts to perform an operation on the latter, control is transferred to a black-box subroutine, which simulates the operation by performing finitely many operations on base objects and returning a value. The correctness property that we consider is **linearizability** [9]. In an execution with operations o_1 and o_2 on the virtual object (**virtual operations** hereafter), the operation o_1 **precedes** the operation o_2 if o_1 returns before o_2 is invoked. An execution is **linearizable** if there exists a total order $\prec$ of virtual operations such that first, if a virtual operation o_1 precedes a virtual operation o_2, then $o_1 \prec o_2$, and second, the return values of the virtual operations are consistent with those obtained by performing the operations in sequence according to the order $\prec$.

3 Queue implementations

David's [4, 5] and Li's [10] implementations can be thought of as variations on Algorithm 1, a simple algorithm in which a single enqueuer writes the enqueued items in order for con-

sumption by a single dequeuer. At the core of both implementations is the idea that either the enqueuers or the dequeuers, but not both, can access the array out of order.

In Li's algorithm, enqueuers divide up the locations in the array with a Fetch&Add object. Because an enqueuer may stall in the interval between reserving a location and writing it, items may be written out of order—an unavoidable consequence of not having a primitive able to achieve consensus among enqueuers. To cope, the dequeuer searches all reserved locations for an item; fortunately, it need not consider locations reserved after the dequeue began. Since the only operations performed by the dequeuer on shared objects are reads, a type of consensus number n allows n dequeuers to simulate a single dequeuer and schedule their dequeue operations on that dequeuer by Herlihy's universal construction.

David's algorithm takes the opposite approach, where the dequeuers divide up the array. Unfortunately, a dequeuer may reserve a location to which the enqueuer has not yet written, in which case we say that the dequeuer has **overtaken** the enqueuer. The simple solutions, where the dequeuer either waits for a value or just returns $\perp$, are not sufficient; the result is an algorithm that is not wait-free or that loses enqueued items.

David's solution to this problem is for the enqueuer to recognize when it has been overtaken and try again in a way that guarantees success. The array of items becomes a two-dimensional array of Swap objects, and dequeuers read locations destructively by swapping in a value $\top$ distinct from the initial value $\perp$. When the enqueuer is overtaken, it swaps out the value $\top$. It is in this case that the second dimension is used: the enqueuer writes the item to the beginning of the next row before informing the dequeuers that this row is now the current one. The dequeuers that reserved empty locations in the previous row return $\perp$, and their operations can be linearized just before the enqueue, when the queue is empty.

There is no straightforward adaptation of David's algorithm to two enqueuers, because with two enqueuers swapping an item into the same location, the second swap may return the item, leaving the enqueuer that performs it unsure as to whether the other swap returns $\top$ or $\perp$. In Algorithm 2, we use a different mechanism for detecting when the enqueuer has been overtaken. Before a dequeuer begins operating on a location (i, j), it writes true to `deqActive`$[i, j]$. When the enqueuer finishes with a location (i, j), it reads `deqActive`$[i, j]$. If the read returns true, the enqueuer assumes that it has been overtaken. This conservative assumption is not always correct, and without further modifications, some items may be returned twice! We add a layer of indirection to address this issue: the two-dimensional array contains **indexes** of items, and the dequeuers use a Fetch&Add object to establish exclusive ownership. A dequeuer that fails to win an item must retry; by retrying in the same row, it turns out that at most two retries are necessary.

Unlike David's algorithm, Algorithm 2 is amenable to an extension of Li's trick. We present the modified enqueue method following the proof of correctness for one enqueuer.

4 Proof of correctness

The main result in this section is the following theorem, which we establish by a sequence of lemmas.

Algorithm 1 Single-enqueuer single-dequeuer queue (folklore)

```
 1: head : integer {enqueuer-local; initially 0}
 2: item : array [0..] of item {initially ⊥}
 3: tail : integer {dequeuer-local; initially 0}

 4: method enq(x : item)
 5:    item[head] := x
 6:    head := head + 1
 7: end method

 8: method deq() : item
 9:    x := item[tail]
10:    if x ≠ ⊥ then
11:       tail := tail + 1
12:    end if
13:    return x
14: end method
```

Theorem 1. *Algorithm 2 is a wait-free linearizable implementation of the type* Queue *for one enqueuer and any number of dequeuers from the types* Fetch&Add *and* Register.

The following lemma implies (bounded) wait-freedom.

Lemma 2. *There is a constant U such that in all executions,* enq *and* deq *operations complete in U steps or less.*

Proof. For the enq method, which has no loops, this is clear. The deq method has one loop, but upon further examination, we find that in the worst case, the loop body executes in its entirety at most twice. If a dequeuer executes the loop body without returning, the local variable `k` is nonzero, and `itemTaken[k].f&a(1)` returns a nonzero value. Another dequeuer, then, must set `k` to the same value and perform `itemTaken[k].f&a(1)` first. Both dequeuers read the value of `k` from locations in the array `itemIndex`, and since each location is accessed by at most one dequeuer, this value is written to two different locations. Any value written to two locations in the array `itemIndex` is the largest written to one row and the smallest written to the next, so it is impossible for a deq operation, which reads values from only one row, to read more than two such values. □

More difficult is showing that Algorithm 2 is linearizable. Any execution that is not linearizable has a finite prefix that is also not linearizable, that is, linearizability is a safety property. Moreover, by wait-freedom, any finite execution has a finite continuation in which processes finish their current queue operations without starting new ones. If the longer execution is linearizable, then so is its prefix, by the same order of operations. It thus suffices to show that any finite execution where all operations finish is linearizable.

Algorithm 2 Single-enqueuer multiple-dequeuer queue

```
 1: deqActive : array [0.., 0..] of boolean {initially false}
 2: enqCount : integer {enqueuer-local; initially 0}
 3: head : integer {enqueuer-local; initially 0}
 4: item : array [1..] of item
 5: itemIndex : array [0.., 0..] of integer {initially 0}
 6: itemTaken : array [1..] of Fetch&Add {initially 0}
 7: row : integer {initially 0}
 8: tail : array [0..] of Fetch&Add {accessed only by dequeuers; initially 0}

 9: method enq(x : item)
10:   enqCount := enqCount + 1
11:   item[enqCount] := x
12:   itemIndex[row, head] := enqCount
13:   if deqActive[row, head] then
14:     itemIndex[row + 1, 0] := enqCount
15:     head := 1
16:     row := row + 1
17:   else
18:     head := head + 1
19:   end if
20: end method

21: method deq() : item
22:   i := row
23:   loop
24:     j := tail[i].f&a(1)
25:     deqActive[i, j] := true
26:     k := itemIndex[i, j]
27:     if k = 0 then
28:       return ⊥
29:     else if itemTaken[k].f&a(1) = 0 then
30:       return item[k]
31:     end if
32:   end loop
33: end method
```

Fix a particular finite execution where, without loss of generality, all operations finish and all enqueued items are distinct. We construct a linearization order $\prec$ as follows. An enq operation e **matches** a deq operation d if e enqueues the item that d dequeues. For deq operations d, let $\mathrm{loc}(d)$ be the last location (i, j) of `itemIndex` read by d. For enq operations e that write exactly one location (i, j) in the array `itemIndex`, let $\mathrm{loc}(e) = (i, j)$. For enq operations e that write two locations (i, j) and $(i + 1, 0)$, there is a unique deq operation d that writes `deqActive`$[i, j]$. Let $\mathrm{loc}(e) = (i, j)$ if e matches d and let $\mathrm{loc}(e) = (i + 1, 0)$ otherwise. For operations o, let $\mathrm{row}(o)$ be the first coordinate of $\mathrm{loc}(o)$.

Lemma 3. *No operation matches more than one other operation.*

Proof. By assumption, no item is enqueued more than once, so no item is written to two locations in the array `item`. In order to return an item `item`$[k]$, a deq operation d must be the first to access `itemTaken`$[k]$, ensuring that d and the enq operation that writes `item`$[k]$ are uniquely matched. □

Lemma 4. *If an* enq *operation e matches a* deq *operation d, then d does not precede e and* $\mathrm{loc}(e) = \mathrm{loc}(d)$.

Proof. The operation d reads the index of the enqueued item from the same location to which e writes that index. Consequently, d cannot precede e, and $\mathrm{loc}(e) = \mathrm{loc}(d)$ by definition. □

For enq operations e, let $\mathrm{orderpt}(e) = \mathrm{line10}(e)$ be the time at which e executes line 10, where the **time** at which a step is taken is the total number of steps that are taken before it. For deq operations d, let $\mathrm{line24}(d)$ be the latest time at which d executes line 24. If d matches an enq operation e, let $\mathrm{orderpt}(d) = \max(\mathrm{line24}(d), \mathrm{line10}(e) + \frac{1}{2})$; otherwise, let $\mathrm{orderpt}(d) = \mathrm{line24}(d)$. For operations o_1 and o_2, write $o_1 \prec o_2$ if $(\mathrm{row}(o_1), \mathrm{orderpt}(o_1)) <_{\mathrm{lex}} (\mathrm{row}(o_2), \mathrm{orderpt}(o_2))$, where the symbol $<_{\mathrm{lex}}$ denotes lexicographic order.

Lemma 5. *The relation $\prec$ is a total order.*

Proof. It suffices to show that the function orderpt is one-to-one. For operations o, either o is unique in taking a step at time $\mathrm{orderpt}(o)$, or o is a deq operation that matches an enq operation e and $\mathrm{orderpt}(o) = \mathrm{line10}(e) + \frac{1}{2}$. In the latter case, no operation $o' \neq o$ satisfies $\mathrm{orderpt}(o') = \mathrm{orderpt}(o)$, since by Lemma 3, the only operation that matches e is o. □

Lemma 6. *If o_1 and o_2 are operations such that o_1 precedes o_2, then $o_1 \prec o_2$.*

Proof. Assume that $o_1 \not\prec o_2$. If $\mathrm{row}(o_1) > \mathrm{row}(o_2)$, then o_1 does not precede o_2, since the value of `row` is nondecreasing. Otherwise, $\mathrm{row}(o_1) = \mathrm{row}(o_2)$ and $\mathrm{orderpt}(o_1) \geq \mathrm{orderpt}(o_2)$. For all operations o, the time $\mathrm{orderpt}(o)$ occurs during o, since either o takes a step at that time, or o is a deq operation that matches an enq operation e, in which case o ends after time $\mathrm{orderpt}(e) = \mathrm{line10}(e)$ by Lemma 4. It follows that o_1 does not precede o_2. □

Lemma 7. *If e_1 and e_2 are* enq *operations, then $e_1 \prec e_2$ if and only if* $\mathrm{loc}(e_1) <_{\mathrm{lex}} \mathrm{loc}(e_2)$. *If d_1 and d_2 are* deq *operations, then* $(\mathrm{row}(d_1), \mathrm{line}24(d_1)) <_{\mathrm{lex}} (\mathrm{row}(d_2), \mathrm{line}24(d_2))$ *if and only if* $\mathrm{loc}(d_1) <_{\mathrm{lex}} \mathrm{loc}(d_2)$.

Proof. There is only one enqueuer, and the pair $(\texttt{row}, \texttt{head})$ increases lexicographically with each enq operation. Line 24 is the invocation of f&a where $\mathrm{loc}(d)$ is obtained. □

Lemma 8. *If d is a* deq *operation, then for all* enq *operations e' with $\mathrm{loc}(e') <_{\mathrm{lex}} \mathrm{loc}(d)$, there exists a* deq *operation d' that matches e'.*

Proof. Fix an enq operation e' with $\mathrm{loc}(e') <_{\mathrm{lex}} \mathrm{loc}(d)$. It suffices to show that some process reads the index written by e', since it follows that some deq operation matches e'. If e' writes exactly one location (i, j) in the array `itemIndex`, then no dequeuer reads that location beforehand, as otherwise the enqueuer would read true from $\texttt{deqActive}[i, j]$. Nevertheless, some deq operation does perform the read. In each row, the set of locations read by dequeuers is a prefix of the row, and some dequeuer reads a location to the right of e'. If $i < \mathrm{row}(d)$, a suitable witness is the deq operation that causes the variable `row` to be incremented; if $i = \mathrm{row}(d)$, a suitable witness is d itself. When e' writes two locations of the array `itemIndex`, the second write necessarily precedes any corresponding read, since it is performed before the enqueuer increments `row`. The remaining arguments parallel the one-write case, with one complication: it may be the case that $\mathrm{loc}(d)$ is between the locations of the first and second write. In this case, $\mathrm{loc}(e') < \mathrm{loc}(d)$ if and only if the deq operation that triggered the second write matches e. □

Lemma 9. *The order $\prec$ is a valid linearization order.*

Proof. Given Lemmas 5 and 6, the only property remaining to be established is that the return values are consistent with the sequential execution determined by the order $\prec$. We prove this by induction on the number of operations.

Specifically, the inductive hypothesis is that through m operations, all return values are correct, and the contents of the queue are the items that have been enqueued but not dequeued, in the order in which they were enqueued. The basis $m = 0$ is trivial. Assuming the inductive hypothesis for m, if the next operation is an enq operation, the inductive hypothesis holds for $m + 1$, since by Lemma 4, enq operations are not preceded by matching deq operations. If the next operation is a deq operation d, then by Lemma 8, every enq operation e' with $\mathrm{loc}(e') <_{\mathrm{lex}} \mathrm{loc}(d)$ has a matching deq operation d'. Each such d' satisfies $\mathrm{line24}(d') < \mathrm{line24}(d)$ by Lemma 7. If d returns $\perp$, then by the definition of $\prec$, it is the case that $e' \prec d$ if and only if $d' \prec d$, so the queue is empty and remains empty. If d matches an enq operation e, then e is the first enq operation not yet matched, by a similar argument. □

We can now prove Theorem 1.

Proof of Theorem 1. Algorithm 2 is wait-free by Lemma 2 and is a linearizable implementation of Queue by Lemma 9. □

Theorem 10. *Algorithm 2 can be implemented by any type of consensus number* 2.

Proof. By the results of Afek, Weisberger, and Weisman [2, 3], any type of consensus number 2 can implement Fetch&Add. □

5 The two-enqueuer case

The two-enqueuer adaptation of Algorithm 2 is presented as Algorithm 4. The main idea is the same as in Li's adaptation, although the details are more complicated: operations by two real processes are scheduled onto one virtual process, which makes progress as long as either real process is active. This scheduling is accomplished by an Agenda object, with a sequential implementation presented as Algorithm 3. Herlihy's universal construction gives a two-process implementation from any type of consensus number 2.

Once an enqueuer schedules an enqueue operation e, it performs the steps that the enqueuer of Algorithm 2 would have up to the point where e is complete. Only finitely many enqueue operations precede e, so this takes only finitely many steps. Exactly once per operation, the enqueue method reads a shared register. To ensure that both enqueuers continue to simulate the same trajectory, they reach consensus on the value of that read.

Theorem 11. *Algorithm 4 is a wait-free linearizable implementation of the type* Queue *for two enqueuers and any number of dequeuers that can be implemented by any type of consensus number* 2*.*

Proof sketch. The new enqueue method is clearly wait-free. Wait-freedom of the new dequeue method and linearizability follow from the fact that each execution of Algorithm 4 begets an execution of Algorithm 2 that has the same collection of enqueue operations, is indistinguishable to the dequeuers, and in which the "real" enqueue operations are active on a super-interval of the corresponding "virtual" enqueue operations. The real enqueuers both take essentially the same steps as the virtual enqueuer, and the virtual enqueuer is deemed to have taken a particular step when it is first taken by a real enqueuer. The construction is made possible by the fact that all of the steps that involve objects shared with the dequeuers are idempotent. There are several categories: reads; enqueuer writes to registers that are written exactly once; and writes to `row`. The latter are idempotent because the values written to `row` increase over time and the dequeuers use only max(`row`). □

Algorithm 3 Agenda object (sequential version)

```
 1: item : array [1..] of item
 2: tail : integer {initially 0}

 3: method append(x : item) : integer
 4:    tail := tail + 1
 5:    item[tail] := x
 6:    return tail
 7: end method

 8: method get(k : integer) : item
 9:    return item[k]
10: end method
```

Algorithm 4 Two-enqueuer multiple-dequeuer queue

```
1: agenda : Agenda {enqueuer-local; initially empty}
2: deqActive : array [0.., 0..] of boolean {initially false}
3: deqActiveRead : array [0.., 0..] of Consensus {enqueuer-local; initially ⊥}
4: enqCount : array [0..1] of integer {enqueuer-local; initially 0}
5: head : array [0..1] of integer {enqueuer-local; initially 0}
6: item : array [1..] of item
7: itemIndex : array [0.., 0..] of integer {initially 0}
8: itemTaken : array [1..] of Fetch&Add {initially 0}
9: row : array [0..1] of integer {initially 0}
10: tail : array [0..] of Fetch&Add {accessed only by dequeuers; initially 0}

11: method enq(x : item)
12:   k := agenda.append(x) {returns the index of x in the agenda}
13:   while enqCount[id] < k do
14:     enqCount[id] := enqCount[id] + 1
15:     item[enqCount[id]] := agenda.get(enqCount[id])
16:     itemIndex[row[id], head[id]] := enqCount[id]
17:     b := deqActive[row[id], head[id]]
18:     if deqActiveRead[row[id], head[id]].decide(b) then
19:       itemIndex[row[id] + 1, 0] := enqCount[id]
20:       head[id] := 1
21:       row[id] := row[id] + 1
22:     else
23:       head[id] := head[id] + 1
24:     end if
25:   end while
26: end method

27: method deq() : item
28:   i := max(row)
29:   loop
30:     j := tail[i].f&a(1)
31:     deqActive[i, j] := true
32:     k := itemIndex[i, j]
33:     if k = 0 then
34:       return ⊥
35:     else if itemTaken[k].f&a(1) = 0 then
36:       return item[k]
37:     end if
38:   end loop
39: end method
```

6 Discussion

Algorithm 4 also works in the **unbounded concurrency** model of Gafni, Merritt, and Taubenfeld [7]. It establishes that two-enqueuer Queue belongs to the unbounded concurrency version of Common2 via the Fetch&Add implementation due to Afek, Gafni, and Morrison [1]. Given the unbounded concurrency Stack by the same authors and a similar adaptation of Li's two-dequeuer Queue, there is currently no set of restrictions for which a bounded concurrency algorithm is known and an unbounded concurrency algorithm is not.

Both our algorithm and Li's require that either the enqueuers or the dequeuers agree on a total order for the items. A general algorithm, if one exists, will have to work in the absence of such an agreement, though we note that the Swap implementation of Afek, Weisberger, and Weisman [3] achieves a similar feat. On the other hand, the implementation of Herlihy and Wing [9] can be modified to be lock-free, so any impossibility result will have to distinguish lock-free implementations from wait-free ones, a property absent from many wait-free impossibility results in the literature.

References

[1] Yehuda Afek, Eli Gafni, and Adam Morrison. Common2 extended to stacks and unbounded concurrency. *Distributed Computing*, 20(4):239–252, 2007.

[2] Yehuda Afek and Eytan Weisberger. The instancy of snapshots and commuting objects. *J. Algorithms*, 30(1):68–105, 1999.

[3] Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A completeness theorem for a class of synchronization objects (extended abstract). In *PODC*, pages 159–170, 1993.

[4] Matei David. A single-enqueuer wait-free queue implementation. In *DISC*, pages 132–143, 2004.

[5] Matei David. Wait-free linearizable queue implementations. Master's thesis, U. Toronto, 2004.

[6] Matei David, Alex Brodsky, and Faith Ellen Fich. Restricted stack implementations. In *DISC*, pages 137–151, 2005.

[7] Eli Gafni, Michael Merritt, and Gadi Taubenfeld. The concurrency hierarchy, and algorithms for unbounded concurrency. In *PODC*, pages 161–169, 2001.

[8] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[9] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[10] Zongpeng Li. Non-blocking implementations of queues in asynchronous distributed shared-memory systems. Master's thesis, U. Toronto, 2001.

[11] Hanan Weisman. Implementing shared memory overwriting objects. Master's thesis, Tel-Aviv U., 1994.